Electronic Notes in Theoretical Computer Science 1 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume1.html 23 pages

Weak Behavioral Subtyping for Types with Mutable Objects

Krishna Kishore Dhara [1] and Gary T. Leavens [1]

Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
dhara@cs.iastate.edu and leavens@cs.iastate.edu

Abstract

This paper studies the question of when one abstract data type (ADT) is a behavioral subtype of another, and proposes a model-theoretic notion of weak behavioral subtyping. Weak behavioral subtyping permits supertype abstraction to be a sound and modular reasoning principle in a language with mutation and limited forms of aliasing. The necessary restrictions on aliasing can be statically checked. Weak behavioral subtyping allows types with mutable objects to be subtypes of types with immutable objects.

1 Introduction

Subtyping is a fundamental semantic concept in object-oriented (OO) languages. In this paper we study *behavioral subtyping*: when one ADT's objects act like those of another. Knowing the conditions on behavioral subtyping is important for guiding the design of ADTs. It is also critical for proving the soundness of logics for OO program verification.

Previous work on the model theory of behavioral subtyping has not allowed mutation and aliasing [3,14] [11,15]. But mutation and aliasing are important in practical OO programming, and many types occurring in practice have objects with mutable (time-varying) state. Although it is possible to imagine an OO language where aliasing is eliminated entirely, existing OO languages do permit aliasing. Unlike Liskov and Wing [18,17], we do not allow arbitrary aliasing, but instead seek a middle ground that permits more behavioral subtype relationships.

The purpose of our study is ultimately to show how to reason in a modular fashion about OO programs. By *modular reasoning*, we mean reasoning such that conclusions about unchanged code remain valid when new behavioral subtypes are added to a program. One modular reasoning technique is

[1] This work was supported in part by the National Science Foundation under Grant CCR-9108654.

supertype abstraction, in which one reasons about the effects of method sends using the properties of the static types of the subexpressions [14,9]. The purpose and justification of a definition of behavioral subtyping is that it makes supertype abstraction sound.

Our technical approach to showing that a definition of behavioral subtyping makes supertype abstraction sound is to capture the conclusions of reasoning via supertype abstraction in a set of expected behaviors. Behaviors that might occur because of subtyping are called *surprising* if they fall outside this set. Thus, showing that a definition of "behavioral subtype" is *adequate* means showing that no surprising behavior is possible when subtyping relationships are required to satisfy the definition.

In this paper we define "weak behavioral subtyping." This definition is weaker than either of Liskov and Wing's definitions [17] because it allows types with mutable objects (hereinafter *mutable types*) to be subtypes of immutable types. We sketch the semantics of a programming language with the necessary aliasing control, and show that weak behavioral subtyping is adequate in the sense described above. Finally we discuss related work and present some conclusions. In this paper, we do not present the model theory of stronger definitions of behavioral subtyping.

2 The Problem

2.1 Reasoning problem with behavioral subtyping and aliasing

The following example motivates reasoning problems with behavioral subtyping. Consider the types `BoolSeq` and `StoreBool` with the following methods. The type `BoolSeq` is a type of boolean sequences, which has only immutable objects. The messages one can send to a `BoolSeq` are the following.

```
method fetch(s: BoolSeq, i: Int): Bool
method update(s: BoolSeq, i: Int, b: Bool): BoolSeq
```

The `update` method produces a new object, which has the same state as the argument `s`, except that in the `i`th position it contains `v`.

The type `StoreBool` has mutable objects. It has the following methods. (A return type of `Void` indicates no result of any useful type is returned.)

```
method fetch(s: BoolStore, i: Int): Bool
method store(s:BoolStore, i:Int, b:Bool): Void
```

There is no subtype relationship between `BoolSeq` and `StoreBool`. Suppose we wish to reason about the part of a program, which we will call an observation, in which the following variables are available.

```
bseq: BoolSeq,  storb: StoreBool,  b: Bool
```

The observation itself consists of a variable declaration and three commands; the declaration names a variable that will be the "output" of the observation. The messages `not` and `equal` have their standard meaning for `Bool` arguments.

```
output: Bool;
```

```
b := fetch(bseq, 2);
store(storb, 2, not(b));
output := equal(b, fetch(bseq, 2))
```

What is the expected set of possible values for `output` in the above observation? The expected set depends on three points:

- Whether one's reasoning technique permits one to assume that identifiers of unrelated types (such as `BoolSeq` and `StoreBool`) cannot be directly aliased.
- Whether `bseq` and `storb` can be aliases for the same object.
- The notion of behavioral subtyping allowed.

These three points are not completely independent. Using Liskov and Wing's definitions of behavioral subtype [18,17], `BoolSeq` and `StoreBool` cannot have a common subtype, because `BoolSeq` objects are immutable (and thus a common subtype would have to violate a history constraint, or would have a mutator corresponding to `store` that could not be explained). So using Liskov and Wing's definition prohibits `bseq` and `storb` from being directly aliased. If `bseq` and `storb` cannot be directly aliased, the set of expected results would be $\{true\}$.

If one's reasoning technique forces one to think about a case where `bseq` and `storb` might be directly aliased, then the set of expected results depends on the notion of behavioral subtyping used. If one had a weaker notion of behavioral subtyping than Liskov and Wing's, then it might be possible for `BoolSeq` and `StoreBool` to have a common subtype with mutable objects. Then `bseq` and `storb` could be directly aliased, and so presumably the set of expected results for the observation above would be $\{true, false\}$. We have not explored such reasoning techniques (which were suggested to us by Ian Maung). However, because objects of type `BoolSeq` are immutable, and the call of a mutator (`store`) is used in the program, it is difficult to imagine the specification of a (most general) common subtype of `BoolSeq` and `StoreBool`. Another problem we see is that, psychologically, programmers would tend to think that because `bseq` is immutable, the only possible result would be *true*, without considering aliasing. Thus such a reasoning technique might be error-prone if used informally. We leave the investigation of such a reasoning technique, and adequate notions of behavioral subtyping for it, as an open problem.

The remaining case is where one's reasoning technique permits one to assume that identifiers of unrelated types cannot be directly aliased. Clearly in this case, such an assumption has to be enforced. If it is, then set of expected results of the observation above is $\{true\}$. However, in this case there is still the possibility that `BoolSeq` and `StoreBool` have a common subtype. Allowing common subtypes, such as `MutableBSeq`, would have a great practical benefit. (That benefit, however, should be weighed against any restrictions on aliasing.)

Thus our problem is twofold: to define a notion of behavioral subtyping that is weaker than Liskov and Wing's, and to state restrictions on aliasing such that it is adequate for reasoning. We refer to our notion of subtyping as "weak behavioral subtyping."

2.2 Reasoning with weak behavioral subtyping

By "reasoning" we mean model-based reasoning with supertype abstraction and with the assumption that identifiers of unrelated types cannot be directly aliased. For the soundness of such reasoning techniques the notion of weak behavioral subtyping should prevent unexpected behavior when subtype objects are manipulated according to specifications of their supertypes.

As an example, suppose we wish to reason about an observation of the following variables.

```
bseq: BoolSeq, b: Bool
```

The observation itself is as follows, where again the declaration gives the output variable for the observation.

```
output: Bool;
b := fetch(bseq, 2);
update(bseq, 2, not(b));
output := equal(fetch(bseq, 2), b);
```

Reasoning at the static types in the above observation, one would obtain the set of expected results, values for `output`, to be $\{true\}$.

Consider a new type `DestructBSeq` which responds to the same set of methods as `BoolSeq`, but such that the method `update` for `DestructBSeq` mutates its first argument. The question is: can `DestructBSeq` be a weak behavioral subtype of `BoolSeq`?

If `DestructBSeq` were to be a weak behavioral subtype of `BoolSeq`, then one could have a state where `bseq` denotes an object of type `DestructBSeq`. The set of results of the above observation in such a state is *false*, which is an unexpected result. This unexpected behavior makes reasoning techniques based on supertype abstraction unsound. Hence `DestructBSeq` cannot be a weak behavioral subtype of `BoolSeq`. We consider soundness of supertype abstraction as an important criteria for defining weak behavioral subtype relations and show a " no surprises" result which guarantees expected behavior.

3 The Language INST and its Semantics

Our model-theoretic approach to solving this problem was described above. To carry out this approach, and to give the reader a concrete picture of the kind of languages to which our results apply, we define an OO programming language and enforce the necessary aliasing constraints in the language. The language used in this paper, INST, is a multimethod language, with an abstract syntax given in Figure 1. The instance variable assignment command ("I_1`.`I_2 `:=` E"), and the object creation ("`new` `I(E*)`") and the instance variable access

Abstract syntax:

P ∈ Program	TD ∈ TypeDecl	T ∈ TypeName	MD ∈ MethDecl
F ∈ Formal	B ∈ Body	M ∈ MainProc	C ∈ Command
E ∈ Expression	D ∈ Decl	A ∈ AliasDecl	

P ::= TD* MD* M
TD ::= `type` I `subtype of` {T*} `instance variables` D `end`
T ::= I
MD ::= `method` I_0 `(` F* `)` `:` T_r A `is` B
F ::= I_1 `:` T
A ::= | `may alias` { I* } `or` { T* }
B ::= D C `return` E
M ::= `main observe` D_1 C_1 `by` D_2 C_2
D ::= | I `:` T `;` D
E ::= N | `nothing` | `true` | `false` | I | I `(` E* `)` | `new` I E* | I_1 `.` I_2
C ::= E | `if` E_1 `then` C_1 `else` C_2 `fi` | I `:=`E | C_1 `;` C_2 | I_1 `.` I_2 `:=` E

Fig. 1. Abstract Syntax of INST. The nonterminal "I" is a variable, and "N" a number. "TD*" is a sequence of zero or more "TD"s (with separators in concrete examples).

("I_1 . I_2") expressions can only be used directly within methods; they cannot be written in the main procedure (M). This provides a simple form of information hiding. For simplicity INST does not distinguish between types and classes. To allow access to the instance variables of method formals, INST do not allow subtype arguments to methods. This is a simplification that avoids treating inheritance. It also would force programmers to define a unique method for each combination of the types of arguments [6].

Figure 2 gives a sample program in INST. For the sake of brevity, we do not present all the method declarations for `BoolSeq`, `StoreBool`, `MutablePair`, and `MutableBSeq`. The method `greater` illustrates the "may alias" construct in INST. The alias component of the method `greater` states that the result is aliased to the second argument `p` or to a variable of type `MutablePair`. More details on aliasing are provided later.

Method dispatching in INST is dynamic in the sense that method lookup does not depend on the static types of variables, but depends on dynamic types of objects. For example, in Fig. 2 the expression `fetch(bseq, 0)` invokes the `fetch` method for `MutableBSeq`, because `bseq` denotes a `MutableBSeq` object, even though the static type of `bseq` is `BoolSeq`. Therefore, INST is a multi-method language [5].

3.1 Denotational Semantics

For various technical reasons, we use a "split semantics" for INST [10]. That is, the meaning of a program is given in two parts: the type and method declarations are compiled into a signature and an algebra over that signature,

```
type BoolSeq subtype of {}
   instance variables fst: Bool; snd: Bool; thd: Bool end;
type StoreBool subtype of {}
   instance variables one: Bool; two: Bool; three: Bool end;
type MutablePair subtype of {}
   instance variables x: Bool; y: Bool end;
type MutableBSeq subtype of {StoreBool, BoolSeq}
   instance variables hd: Bool; tl: MutablePair end;
method mkBoolSeq(): EmptySeq
   is bseq: BoolSeq;
      bseq := new BoolSeq; set_fst(bseq, false);
      set_snd(bseq, false); set_thd(bseq, false);
      return bseq;
method fetch(s: BoolSeq, i: Int):Bool
   is result: Bool;
      if equal(i, 0) then result := s.fst
         else if equal(i,1) then result := s.snd
                 else result := s.thd fi fi
      return result;
method mkMutableBseq(): MutableBseq
   is mb: MutableBSeq; p: MutablePair;
      mb := new MutableBseq; p := new MutablePair;
      set_x(p, false); set_y(p, false);
      set_hd(mb, false); set_tl(mb, p);
      return mb;
method greater(mb: MutableBSeq; p: MutablePair): MutablePair
                     may alias {p} or {Pair}
   is result: Pair;
      if less(mb.tl, p) then result := p else result := mb.tl fi
      return result;
...
...
main observe
       bseq: BoolSeq; b: Bool
       bseq := mkMutableBSeq();
       b := fetch(bseq, 0)
     by
       output: Bool;
       update(bseq, 0, not(b));
       output := equal(fetch(bseq, 0), b)
```

Fig. 2. Part of a sample program in INST. The set of expected results, the possible values for output, should be {true}.

and the meaning of the main procedure uses conventional denotational techniques. The meaning functions for declarations, commands, and expressions take algebra as an argument.

For purposes of this paper, in which we define observations that may observe states over algebras, the main procedure (M) has a strange syntax. It consists of two sequences of declarations and commands. The reason for splitting the main procedure in this way is to indicate in what part supertype abstraction is used. Supertype abstraction would be used to reason about the part of the main procedure following the keyword by, which thus defines an observation of the state constructed by the first part. The meaning of the second part is technically a function from algebras to observations of states over algebras. To get the results of a program, one passes the algebra and state, constructed by the declarations and the first part of the main procedure, to the observation obtained by the second part of the main procedure.

The semantics of a program is shown formally below. Most of the notation has not been discussed yet, but it seemed helpful to show the valuation function for programs before launching into the details. Nonstandard notations not explained in this paragraph will be explained further below. The signature Σ^{INST} and the Σ^{INST}-algebra $\mathbf{A}^{\mathrm{INST}}$ give the signature and semantics of the visible types (see Figures 4 and 6 in [10]). The valuation function for type declaration sequences, $\mathcal{TD}*$, adds to the signature and algebra primitive operations for each type declared; these primitive operations are used by the semantics of expressions and commands for creating objects and for accessing their instance variables. Once $\mathcal{MD}*$ has processed all the method declarations, these primitive methods are suppressed. A signature without the primitive operation symbols is produced by *hideInternalMessages*. The notation $\mathbf{A}'|_{(\mathit{hideInternalMessages}\ \Sigma')}$ is the reduct of $\mathbf{A}'$ without these primitives.

$$\mathcal{P} : \mathrm{Program} \to (\mathit{SIGS} \times \mathit{ALG} \times \mathit{TENV} \times \mathit{STATE} \times (\mathit{ALG} \to \mathit{OBS}))_\perp$$

$$\begin{aligned}
&\mathcal{P}[\![\mathrm{TD}^*\ \mathrm{MD}^*\ \mathrm{M}]\!] = \\
&\quad \textbf{let } (\Sigma, \mathbf{A}) = \mathcal{TD}*[\![\mathrm{TD}^*]\!]\ \Sigma^{\mathrm{INST}}\ \mathbf{A}^{\mathrm{INST}} \textbf{ in} \\
&\quad \textbf{let } (\Sigma', \mathbf{A}') = \mathcal{MD}*[\![\mathrm{MD}^*]\!]\ \Sigma\ \mathbf{A} \textbf{ in} \\
&\quad \textbf{let } (\Sigma'', \mathbf{A}'') = (\mathit{hideInternalMessages}\ \Sigma',\ A'|_{(\mathit{hideInternalMessages}\ \Sigma')}) \textbf{ in} \\
&\quad \textbf{let } (H, s_1, f) = \mathcal{M}''_\Sigma[\![\mathrm{M}]\!]\ \mathbf{A}'' \textbf{ in } \ (\Sigma'', \mathbf{A}'', H, s_1, f)
\end{aligned}$$

Due to lack of space we do not give the details of the semantics of type and method declarations. Instead, we define the signatures and algebras that they denote, and then turn to the semantics of expressions, declarations, commands, and the main procedure.

To define observations, we fix a set of the visible (or built-in) types, $\mathit{VIS} = \{\mathtt{Int}, \mathtt{Bool}\}$. The externally visible values of these types are:

$$\mathit{EXTERNALS}_{\mathtt{Int}} \stackrel{\mathrm{def}}{=} \{0, 1, -1, \cdots\} \quad \text{and} \quad \mathit{EXTERNALS}_{\mathtt{Bool}} \stackrel{\mathrm{def}}{=} \{\mathit{true}, \mathit{false}\}.$$

Signatures are roughly as in Reynolds's category sorted algebras [19], with the addition of information about aliasing that is used in our static restrictions

on aliasing.

Definition 3.1 [*SIGS*, **signature**] The set *SIGS*, consists of all *signatures*, Σ, which are tuples $(TYPES, \leq, OPS, ResType, RetAlias)$ such that:

- $TYPES$ is a set of type symbols such that $VIS \subseteq TYPES$ and `Void` $\in TYPES$.
- $\leq$ is a preorder on $TYPES$, such that if $S \leq T$ and $T \in VIS$, then $S = T$.
- OPS is a family of sets of operation symbols, indexed by the natural numbers,
- $ResType$ is a family of partial functions indexed by the natural numbers, such that for each natural number n, $ResType_n : OPS_n \times TYPES^n \rightarrow TYPES_\perp$, and $ResType$ is monotone. That is, for all $g \in OPS$, and for all tuples of types $\vec{S} \leq \vec{T}$, if $ResType(g, \vec{T}) \neq \perp$ then $ResType(g, \vec{S}) \neq \perp$ and $ResType(g, \vec{S}) \leq ResType(g, \vec{T})$.
- $RetAlias$ limits the types of variables that may be directly aliased to the result of a method (based on the types of variables aliased to the actuals). $RetAlias$ is a family of partial functions indexed by the natural numbers, such that for each natural number n, $RetAlias_n : OPS_n \times TYPES^n \rightarrow (PowerSet(TYPES)^n \rightarrow PowerSet(TYPES)_\perp)$.

To simplify notation we usually write $g \in OPS$ as shorthand for $g \in \bigcup_{n \in Nat} OPS_n$. Similarly we write $ResType$ for $ResType_n$ and $RetAlias$ for $RetAlias_n$.

As an example Fig. 3 gives a part of the signature for the INST program in Fig. 2. The $RetAlias$ function in Fig. 3 gives the alias relation between the arguments and the result for each operation. Recall that the method `greater` returns either a direct alias to its second argument or a direct alias to an instance variable of type `MutablePair`. This is captured in the $RetAlias$ function for `greater`.

Our models of abstract types with mutable objects are algebraic [20,7,10]. Objects are modeled by typed locations containing values, which may in turn contain locations. We define algebras and stores simultaneously, because the operations of an algebra take and return a store [10].

Definition 3.2 [$ALG(\Sigma)$, Σ-**algebra,** $STORE$] The set $ALG(\Sigma)$, consists of all Σ-*algebras*,

$$\mathbf{A} = (SORTS^{\mathbf{A}}, LOCS^{\mathbf{A}}, ObjectTypes^{\mathbf{A}}, VALS^{\mathbf{A}}, TtoS^{\mathbf{A}}, OPS^{\mathbf{A}}, externVal^{\mathbf{A}}),$$

such that:

- $SORTS^{\mathbf{A}} \supseteq TYPES$ is a set of sort symbols,
- $LOCS^{\mathbf{A}}$ is a family of sets, indexed by $ObjectTypes^{\mathbf{A}}$, representing typed locations,
- $ObjectTypes^{\mathbf{A}} \subseteq SORTS^{\mathbf{A}}$ is the set of object type symbols,

$$
\begin{aligned}
TYPES &\stackrel{\text{def}}{=} \{\texttt{Bool}, \texttt{Int}, \texttt{Void}, \texttt{BoolSeq}, \texttt{StoreBool}, \texttt{MutablePair}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{MutableBSeq}\} \\
\leq \quad &\stackrel{\text{def}}{=} \{(T,T) | T \in TYPES\}\{(\texttt{MutableBSeq}, \texttt{BoolSeq}), \\
&\qquad\qquad\qquad\qquad (\texttt{MutableBSeq}, \texttt{StoreBool})\} \\
OPS \quad &\stackrel{\text{def}}{=} \{\texttt{true}, \cdots, \texttt{fetch}, \cdots, \texttt{greater}, \cdots\}
\end{aligned}
$$

ResType

$$
\begin{aligned}
&ResType(\texttt{true}, ()) &&= \texttt{Bool} \\
&ResType(\texttt{fetch}, (\texttt{BoolSeq}, \texttt{Int})) &&= \texttt{Bool} \\
&ResType(\texttt{fetch}, (\texttt{MutableBSeq}, \texttt{Int})) &&= \texttt{Bool} \\
&ResType(\texttt{greater}, (\texttt{MutableBSeq}, \texttt{MutablePair})) &&= \texttt{MutablePair} \\
&\cdots
\end{aligned}
$$

RetAlias

$$
\begin{aligned}
&RetAlias(\texttt{true}, ()) &&= \lambda().\{\} \\
&RetAlias(\texttt{fetch}, (\texttt{BoolSeq}, \texttt{Int})) &&= \lambda(arg1, arg2).\{\} \\
&RetAlias(\texttt{fetch}, (\texttt{MutableBSeq}, \texttt{Int})) &&= \lambda(arg1, arg2).\{\} \\
&RetAlias(\texttt{greater}, (\texttt{MutableBSeq}, \texttt{MutablePair})) &&= \lambda(arg1, arg2). \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad arg2 \cup \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\texttt{MutablePair}\} \\
&\cdots
\end{aligned}
$$

Fig. 3. Part of the signature, Σ', for the INST program given in Figure 2

- $VALS^{\mathbf{A}}$ is a family of abstract values indexed by $SORTS^{\mathbf{A}}$, such that for each $T \in ObjectTypes^{\mathbf{A}}$, $VALS^{\mathbf{A}}_T = LOCS^{\mathbf{A}}_T$
- $TtoS^{\mathbf{A}} : ObjectTypes^{\mathbf{A}} \to SORTS^{\mathbf{A}}$ is a function that gives a sort symbol for each object type symbol,
- $OPS^{\mathbf{A}}$ is a family of operation interpretations indexed by the natural numbers, such that for each $n \in Nat$ and $g \in OPS_n$, there is a polymorphic partial function $g^{\mathbf{A}} \in OPS^{\mathbf{A}}_n$ where for each $\vec{S} \in TYPES^n$ and $T \in TYPES$, if $ResType(g, \vec{S}) = T$ then $g^{\mathbf{A}}$ satisfies $g^{\mathbf{A}} : (VALS^{\mathbf{A}}_{\vec{S}} \times STORE[\mathbf{A}]) \to ((\cup_{U \leq T} VALS^{\mathbf{A}}_T) \times STORE[\mathbf{A}])_\perp$,
- $externVal^{\mathbf{A}}$ is a family of functions indexed by VIS, such that for each $T \in VIS$, $externVal^{\mathbf{A}}_T : VALS^{\mathbf{A}}_T \times STORE[\mathbf{A}] \to (EXTERNALS_T)_\perp$,

and $STORE[\mathbf{A}] \stackrel{\text{def}}{=} LOCS^{\mathbf{A}} \stackrel{\text{fin}}{\to} VALS^{\mathbf{A}}$ is such that if $\sigma : STORE[\mathbf{A}]$ and $l \in LOCS^{\mathbf{A}}_T \cap dom(\sigma)$, then $\sigma(l) \in \cup_{U \leq T} VALS^{\mathbf{A}}_{TtoS^{\mathbf{A}}[U]}$.

$$SORTS^{\mathbf{A}'} \stackrel{\text{def}}{=} TYPES \bigcup \{\text{Var[T]} \mid T \in TYPES\} \bigcup$$
$$\{\text{sortFor(}\texttt{BoolSeq}\text{)}, \text{sortFor(}\texttt{StoreBool}\text{)},$$
$$\text{sortFor(}\texttt{MutablePair}\text{)}, \text{sortFor(}\texttt{MutableBSeq}\text{)}\}$$

$$ObjectTypes^{\mathbf{A}'} \stackrel{\text{def}}{=} \{\text{Var[T]} \mid T \in TYPES\} \bigcup \{\texttt{BoolSeq}, \texttt{MutableBSeq}$$
$$\texttt{StoreBool}, \texttt{MutablePair}\}$$

$$LOCS^{\mathbf{A}'}_T \stackrel{\text{def}}{=} \{l^T_i \mid i \in Nat\}, \quad \text{for each } T \in ObjectTypes^{\mathbf{A}'}$$

$$VALS^{\mathbf{A}'}_{\texttt{Int}} \stackrel{\text{def}}{=} VALS^{\mathbf{A}'}_{\texttt{Int}}$$

$$VALS^{\mathbf{A}'}_{\text{sortFor(}\texttt{BoolSeq}\text{)}} \stackrel{\text{def}}{=} \{(v_{fst}, v_{snd}, v_{thd}) \mid v_{fst}, v_{snd}, v_{thd} \in VALS^{\mathbf{A}'}_{\texttt{Bool}}\}$$

$\ldots$

$$VALS^{\mathbf{A}'}_{\text{sortFor(}\texttt{MutableBSeq}\text{)}} \stackrel{\text{def}}{=} \{(v_h, l_t) \mid v_h \in VALS^{\mathbf{A}'}_{\texttt{Bool}}, l_t \in LOCS^{\mathbf{A}'}_{\texttt{MutablePair}}\}$$

Type to Sort Mapping ($TtoS^{\mathbf{A}'}$)

Var[T]	$\mapsto T, \forall T \in TYPES$
`BoolSeq`	$\mapsto$ sortFor(`BoolSeq`)
`StoreBool`	$\mapsto$ sortFor(`StoreBool`)
`MutablePair`	$\mapsto$ sortFor(`MutablePair`)
`MutableBSeq`	$\mapsto$ sortFor(`MutableBSeq`)

$$externVal^{\mathbf{A}'}$$

$$externVal^{\texttt{Bool}}_{\mathbf{A}'}(v, \sigma) \stackrel{\text{def}}{=} v$$

$$externVal^{\texttt{Int}}_{\mathbf{A}'}(v, \sigma) \stackrel{\text{def}}{=} v$$

Fig. 4. Components (part 1) of a Σ'-algebra, $\mathbf{A}'$, for the INST program in Fig. 2.

We write $l : T$ as an abbreviation for $l \in LOCS^{\mathbf{A}}_T$. Fig. 4 and Fig. 5 give part of the algebra, $\mathbf{A}'$, corresponding to the program in Fig. 2.

The set $TENV(\Sigma)$ of *type environments* over a signature Σ is defined by $TENV(\Sigma) = \text{Identifier} \stackrel{\text{fin}}{\to} TYPES$. Let H stand for a type environment below.

A state consists of an environment and a store. The set $ENV_H[\mathbf{A}]$ of *H-environments over* $\mathbf{A}$ is the set of all mappings, $\eta : \text{Identifier} \stackrel{\text{fin}}{\to} LOCS^{\mathbf{A}}$, such that for every $T \in TYPES$, if $H(x) = T$ then $x \in dom(\eta)$ and $\eta(x) \in LOCS^{\mathbf{A}}_T$. The set, $STATE_H[\mathbf{A}]$, of *H-states over* $\mathbf{A}$ is defined by $STATE_H[\mathbf{A}] \stackrel{\text{def}}{=} ENV_H[\mathbf{A}] \times STORE[\mathbf{A}]$. We write $ENV[\mathbf{A}]$ for $\cup_{H \in TENV(\Sigma)} ENV[\mathbf{A}]$, and $STATE[\mathbf{A}]$ for $ENV[\mathbf{A}] \times STORE[\mathbf{A}]$.

$$OPS^{\mathbf{A}'}$$

$$\begin{array}{lcl}
\texttt{add}^{\mathbf{A}'}((v_1, v_2), \sigma) & \stackrel{\text{def}}{=} & (v_1 + v_2, \sigma) \\
\texttt{fetch}^{\mathbf{A}'}((l^{\texttt{BoolSeq}}, v^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (f, s, t) = (\sigma\ l^{\texttt{BoolSeq}}) \textbf{ in} \\
 & & \textbf{if } \texttt{equal}^{\mathbf{A}'}(v^{\texttt{Int}}, 0) \textbf{ then } (f, \sigma) \\
 & & \textbf{else if } \texttt{equal}^{\mathbf{A}'}(v^{\texttt{Int}}, 0) \\
 & & \quad \textbf{then } (s, \sigma) \\
 & & \quad \textbf{else } (t, store) \\
\texttt{fetch}^{\mathbf{A}'}((l^{\texttt{MutableBSeq}}, v^{\texttt{Int}}), \sigma) & \stackrel{\text{def}}{=} & \textbf{let } (fst, l) = (\sigma\ l^{\texttt{BoolSeq}}) \textbf{ in} \\
 & & \textbf{if } \texttt{equal}^{\mathbf{A}'}(v^{\texttt{Int}}, 0) \textbf{ then } (fst, \sigma) \\
 & & \textbf{else if } \texttt{equal}^{\mathbf{A}'}(v^{\texttt{Int}}, 0) \\
 & & \quad \textbf{then } \texttt{fst}^{\mathbf{A}'}(l, \sigma) \\
 & & \quad \textbf{else } \texttt{snd}^{\mathbf{A}'}(l, \sigma) \\
\ldots & &
\end{array}$$

Fig. 5. Components (part 2) of a Σ'-algebra, $\mathbf{A}'$, for the INST program in Fig. 2.

Definition 3.3 [nominal state] A H-state (η, σ) is *nominal* if and only if σ is nominal. A store $\sigma \in STORE[\mathbf{A}]$ is *nominal* if and only if for all locations $l : T \in dom(\sigma)$, $\sigma(l) \in VALS_T^{\mathbf{A}}$.

The main procedure (M) returns a type environment, a state, and a function from algebras to observations. This function is defined by the second half of the main procedure. An observation takes a state, such as the one produced by the first half of the main procedure, and "prints" the values of the variables in D_2. *H-observations* are defined as follows.

$$OBS_H[\mathbf{A}] \stackrel{\text{def}}{=} STATE_H[\mathbf{A}] \to ANSWERS_\perp \tag{1}$$

$$ANSWERS \stackrel{\text{def}}{=} \text{Identifier} \stackrel{\text{fin}}{\to} EXTERNALS \tag{2}$$

The variables declared in D_2 must have visible type. This condition is checked by *typeEnvAndCheckVisible*, which produces a type environment if they are visible (and $\perp$ otherwise).

To simplify notation we omit Σ from $ALG(\Sigma)$ and we write ALG for the family $\cup_{\Sigma \in SIGS} ALG(\Sigma)$. Similarly we use $TENV$ for $TENV(\Sigma)$, $STORE$ for $STORE[\mathbf{A}]$, ENV for $ENV_H[\mathbf{A}]$, $STATE$ for $STATE_H[\mathbf{A}]$, and OBS for $OBS_H[\mathbf{A}]$. However for a type like the $\mathcal{M}$ below it should be understood that if Σ is the signature passed to $\mathcal{M}$, then the algebra $\mathbf{A}$ passed to $\mathcal{M}$ must be a Σ-algebra. Similarly, the type environments, states, and observations will also match.

$\mathcal{M} : SIGS \to \text{MainProc} \to ALG \to (TENV \times STATE \times (ALG \to OBS))_\perp$
$\mathcal{M}_\Sigma[\![\ \texttt{main observe}\ \mathrm{D}_1\ \mathrm{C}_1\ \texttt{by}\ \mathrm{C}_2\ \mathrm{D}_2]\!]\ \mathbf{A} =$

$$
\begin{array}{l}
\textbf{let } H = \mathit{typeEnv}[\![\mathrm{D}_1]\!] \textbf{ in} \\
\textbf{let } (\eta, \sigma) = \mathcal{D}_\Sigma[\![\mathrm{D}_1]\!]\ \mathbf{A}\ (\mathit{emptyEnviron}, \mathit{emptyStore}) \textbf{ in} \\
\textbf{let } \sigma' = \mathcal{C}_\Sigma[\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in} \\
\textbf{let } H' = \mathit{typeEnvAndCheckVisible}[\![\mathrm{D}_2]\!] \textbf{ in} \\
\textbf{let } f\ =\ (\lambda \mathbf{B}\,.\,\lambda(\eta_\mathbf{B}, \sigma_\mathbf{B})\,. \\
\qquad\qquad \textbf{let } \sigma'_\mathbf{B} = \mathcal{C}_\Sigma[\![\mathrm{C}_2]\!]\ B\ (\eta_\mathbf{B}, \sigma_\mathbf{B}) \textbf{ in} \\
\qquad\qquad \textbf{let } (\eta''_\mathbf{B}, \sigma''_\mathbf{B}) = \mathcal{D}_\Sigma[\![\mathrm{D}_2]\!]\ \mathbf{B}\ (\eta_\mathbf{B}, \sigma'_\mathbf{B}) \textbf{ in} \\
\qquad\qquad \lambda[\![I]\!]\,.\,\textbf{let } T = H'[\![I]\!] \textbf{ in } \mathit{externVal}^\mathbf{B}_T(\eta''_\mathbf{B}[\![I]\!], \sigma''_\mathbf{B})) \\
\textbf{in } (H,\ (\eta, \sigma'),\ f)
\end{array}
$$

For a given signature, Σ, an expression has a meaning which depends on a Σ-algebra. We do not show the semantics for the expressions of the form "`new I(E*)`" or "$\mathrm{I}_1\,.\,\mathrm{I}_2$", because these cannot occur in the main procedure, and so play no role in defining observations.

$$
\begin{array}{l}
\mathcal{E} : \mathit{SIGS} \to \text{Expression} \to \mathit{ALG} \to \mathit{STATE} \to (\mathit{VALS} \times \mathit{STORE})_\perp \\
\mathcal{E}_\Sigma[\![\mathrm{N}]\!]\ \mathbf{A}\ (\eta, \sigma) = \mathcal{N}_\Sigma[\![\mathrm{N}]\!]\ \mathbf{A}\ \sigma \\
\mathcal{E}_\Sigma[\![\texttt{nothing}]\!]\ \mathbf{A}\ (\eta, \sigma) = \texttt{nothing}^\mathbf{A}((), \sigma) \\
\mathcal{E}_\Sigma[\![\texttt{true}]\!]\ \mathbf{A}\ (\eta, \sigma) = \texttt{true}^\mathbf{A}((), \sigma) \\
\mathcal{E}_\Sigma[\![\texttt{false}]\!]\ \mathbf{A}\ (\eta, \sigma) = \texttt{false}^\mathbf{A}((), \sigma) \\
\mathcal{E}_\Sigma[\![\mathrm{I}]\!]\ \mathbf{A}\ (\eta, \sigma) = (\textbf{let } v = (\sigma\ (\eta[\![\mathrm{I}]\!])) \textbf{ in } (v, \sigma)) \\
\mathcal{E}_\Sigma[\![\mathrm{I}(\hat{\mathrm{E}})]\!]\ \mathbf{A}\ (\eta, \sigma) = \textbf{let } (\hat{v}, \sigma') = \mathcal{E}*_\Sigma[\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in } \ \mathrm{I}^\mathbf{A}(\mathit{productize}\ \hat{v}, \sigma') \\
\mathcal{E}* : \mathit{SIGS} \to \text{Expression-List} \to \mathit{ALG} \to \mathit{STATE} \\
\qquad\qquad \to (\mathit{List}(\mathit{VALS}^\mathbf{A}) \times \mathit{STORE}[\mathbf{A}])_\perp \\
\mathcal{E}*_\Sigma[\![]\!]\ \mathbf{A}\ (\eta, \sigma) = (\mathit{nil}, \sigma) \\
\mathcal{E}*_\Sigma[\![\hat{\mathrm{E}}\ \mathrm{E}_n]\!]\ \mathbf{A}\ (\eta, \sigma) = \textbf{let } (\hat{v}, \sigma') = \mathcal{E}*_\Sigma[\![\hat{\mathrm{E}}]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } (v_n, \sigma_n) = \mathcal{E}_\Sigma[\![\mathrm{E}_n]\!]\ \mathbf{A}\ (\eta, \sigma') \textbf{ in} \\
\qquad\qquad\qquad\qquad ((\mathit{addToEnd}\ \hat{v}\ v_n),\ \sigma_n)
\end{array}
$$

The semantics of commands is straightforward. Assignment binds variable locations to objects or values.

$$
\begin{array}{l}
\mathcal{C} : \mathit{SIGS} \to \text{Command} \to \mathit{ALG} \to \mathit{STATE} \to \mathit{STORE}_\perp \\
\mathcal{C}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta, \sigma) = \textbf{let } (v, \sigma') = \mathcal{E}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in } \ \sigma' \\
\mathcal{C}_\Sigma[\![\mathrm{C}_1\,;\ \mathrm{C}_2]\!]\ \mathbf{A}\ (\eta, \sigma) = \textbf{let } \sigma_1 = \mathcal{C}_\Sigma[\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in } \ \mathcal{C}_\Sigma[\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta, \sigma_1) \\
\mathcal{C}_\Sigma[\![\texttt{if } \mathrm{E}_1 \texttt{ then } \mathrm{C}_1 \texttt{ else } \mathrm{C}_2 \texttt{ fi}]\!]\ \mathbf{A}\ (\eta, \sigma) = \\
\qquad \textbf{let } (v, \sigma') = \mathcal{E}_\Sigma[\![\mathrm{E}_1]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in} \\
\qquad \textbf{if } \mathit{externVal}^\mathbf{A}_{\mathrm{Bool}}(v, \sigma') \textbf{ then } (\mathcal{C}_\Sigma[\![\mathrm{C}_1]\!]\ \mathbf{A}\ (\eta, \sigma')) \textbf{ else } (\mathcal{C}_\Sigma[\![\mathrm{C}_2]\!]\ \mathbf{A}\ (\eta, \sigma')) \\
\mathcal{C}_\Sigma[\![\mathrm{I} := \mathrm{E}]\!]\ \mathbf{A}\ (\eta, \sigma) = \\
\qquad \textbf{let } H \text{ be such that } (\eta, \sigma) \in \mathit{STATE}_H[\mathbf{A}] \textbf{ in} \\
\qquad \textbf{let } (v, \sigma') = \mathcal{E}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta, \sigma) \textbf{ in} \\
\qquad \textbf{if } v \notin \bigcup_{U \le H(v)} \mathit{VALS}^\mathbf{A}_U \textbf{ then } \perp \textbf{ else } [(\eta\ [\![I]\!]) \mapsto v]\sigma'
\end{array}
$$

Declarations bind variables to variable locations. The $\mathit{nextFree}[T]$ function in the meaning of a declaration returns the next free location of type T in a given store.

$$
\begin{array}{l}
\mathcal{D} : \mathit{SIGS} \to \text{Decl} \to \mathit{ALG} \to \mathit{STATE} \to \mathit{STATE}_\perp \\
\mathcal{D}_\Sigma[\![]\!]\ \mathbf{A}\ s = s
\end{array}
$$

$$\mathcal{D}_\Sigma[\![\mathrm{I}\colon \mathrm{T}]\!]\ \mathbf{A}\ (\eta,\sigma) = \textbf{let}\ T' = \mathcal{T}_\Sigma[\![\mathrm{T}]\!]\ \textbf{in}\ \textbf{let}\ l = \mathit{nextFree}[\mathrm{Var}[\mathrm{T}']](\sigma)\ \textbf{in}\ ([\mathrm{I} \mapsto l]\eta, \sigma)$$
$$\mathcal{D}_\Sigma[\![\mathrm{D}_1\ ;\ \mathrm{D}_2]\!]\ \mathbf{A}\ s = \mathcal{D}_\Sigma[\![\mathrm{D}_2]\!]\ \mathbf{A}\ (\mathcal{D}_\Sigma[\![\mathrm{D}_1]\!]\ \mathbf{A}\ s)$$

The semantics of bodies (of methods) will also be used to help define the restrictions on aliasing. The type of a body B is the type of the return expression.

$$\mathcal{B} : \mathit{SIGS} \to \mathrm{Body} \to \mathit{ALG}) \to \mathit{STATE} \to (\mathit{VALS} \times \mathit{STORE})_\perp$$
$$\mathcal{B}_\Sigma[\![\mathrm{D\ C\ \texttt{return}\ E}]\!]\ \mathbf{A}\ s = \textbf{let}\ (\eta_1,\sigma_1) = \mathcal{D}_\Sigma[\![\mathrm{D}]\!]\ \mathbf{A}\ s\ \textbf{in}\ \textbf{let}\ \sigma_2 = \mathcal{C}_\Sigma[\![\mathrm{C}]\!]\ \mathbf{A}\ (\eta_1,\sigma_1)\ \textbf{in}\ \ \mathcal{E}_\Sigma[\![\mathrm{E}]\!]\ \mathbf{A}\ (\eta_1,\sigma_2)$$

3.2 Enforcing Restrictions on Aliasing

For our notion of weak behavioral subtyping to be adequate, we need to prevent direct aliasing between related, but distinct, types. Since we also want to be able to reason modularly, we need to also prevent direct aliasing between variables of unrelated types, because two unrelated types might, at some later time, have a common subtype. Thus, in this section, we define restrictions on aliasing such that variables of different types cannot be directly aliased. We do this by an abstract interpretation of INST programs, which conservatively estimates the set of types that may be aliased to each expression result. This set of types is called an *alias type set*.

The definitions below are for declarations, commands, and expressions that could be executed in the main procedure. For this purpose we define the set "MBody" as the subset of Body that includes only declarations, commands, and expressions that can be written in the main procedure.

A location l is *reachable* in a H-state s over an Σ-algebra $\mathbf{A}$ if and only if there exists a type T, and a body $B \in \mathrm{MBody}$ such that $\Sigma; H \vdash B : T$ and $(l,\sigma) = \mathcal{B}_\Sigma[\![B]\!]\ \mathbf{A}\ s$. The notation "$\Sigma; H \vdash B : T$" means that for Σ and H, B can be proved to have type T (using the rules in Fig. 6).

The alias type set of a location in a H-state over an algebra is defined by the following.

$$\mathit{aliasTypeSet}_\Sigma(H, \mathbf{A}, l, s) \stackrel{\text{def}}{=} \{T \mid T \in \mathit{TYPES},\ B \in \mathrm{MBody},\ \Sigma; H \vdash B : T,\ (l,\sigma) = \mathcal{B}_\Sigma[\![B]\!]\ \mathbf{A}\ s\}. \tag{3}$$

Since the location returned by `mkMutableBSeq()` in the main procedure in Fig. 2 is not reachable, its alias type set is `{}`. However, after executing the assignment command, `bseq := mkMutableBSeq()`, the alias type set of the location will be `{BoolSeq}`, because that is the static type of the variable `bseq`.

If a location's alias type set contains at most one type, then it can only be aliased by variables of the same type. This property is captured by the following.

$$\mathit{subtypeCheck}(r) \stackrel{\text{def}}{=} (S \in r \wedge T \in r) \Rightarrow (S = T) \tag{4}$$

Alias legality means that every reachable location has this property.

Definition 3.4 [**alias legality,** $stAliasOk_\Sigma$] Let $\mathbf{A}$ be a Σ-algebra, H be a $TENV$, and $s \in STATE_H[\mathbf{A}]$. Then s is said to be *alias legal*, written $stAliasOk_\Sigma(H, \mathbf{A}, s)$, if and only if for all reachable locations l in s, $subtypeCheck(aliasTypeSet_\Sigma(H, \mathbf{A}, l, s))$.

Figure 6 gives the type and alias checking rules for expressions, declarations, and commands that can appear in the main procedure. For expressions, the notation $\Sigma; H \vdash E : T :: r$ means E has static type T and r is an upper bound on the alias type set of the result of E. The rule for the assignment statement checks that the binding does not produce illegal aliasing. For declarations, $\Sigma; H \vdash D \Longrightarrow H'$ means H' is the type environment after elaborating D. For methods the alias type set of the result is declared, and compiled into the signature in its $RetAlias$. We do not give the exact rules for method bodies, because they are not needed in this paper.

To see the practical implications of our technique for restricting aliasing, it is useful to consider how the property that variables of distinct types are not directly aliased would be established in the body of a method, after binding actuals to formals. One option would be to prohibit any direct aliasing among the actuals in a call. This is more restrictive than we need, because aliasing between formals of the same type is not a problem. Instead, we require that the programmer write enough methods so that any call with directly aliased actuals will be handled by a method implementation where the formals corresponding to those actuals have the same type. For example, consider a method `foo` with two arguments. If the same object is to be passed for both arguments to `foo` then the call to `foo` will be handled by a method which has two formals of the same type as the dynamic type of the object. In a more realistic language with method inheritance, this would force the programmer to write specialized versions of `foo` having both arguments of the same type, some of which might not otherwise have to be written.

Because we do not work with methods in this paper, and because we work with algebras that may not result from INST programs, we need to impose an equivalent condition that calling an operation in a Σ-algebra cannot result in illegal aliases. To prevent illegal aliases in the result state, the H-state, s, that results from a call to $g^{\mathbf{A}}$ must satisfy $stAliasOk_\Sigma(H, \mathbf{A}, s)$. To prevent the result itself from being directly aliased with variables of different types, the actual alias type set of the result must be smaller than that declared.

Definition 3.5 [**preserves alias legality**] Let $\mathbf{A}$ be a Σ-algebra. Let H be a type environment. Then $\mathbf{A}$ *preserves alias legality* if and only if for each H-state (η, σ) such that $stAliasOk_\Sigma(H, \mathbf{A}, (\eta, \sigma))$, for each operation $g \in OPS$, for each tuple of types $\vec{S}$, if $RetAlias(g, \vec{S}, \vec{r}) = \hat{r}$, $\vec{v} \in VALS^{\mathbf{A}}_{\vec{S}}$, and $(l, \sigma') = g^{\mathbf{A}}(\vec{v}, \sigma)$, then:

$$stAliasOk_\Sigma(H, \mathbf{A}, (\eta, \sigma')) \wedge (l \in LOCS^{\mathbf{A}} \Rightarrow aliasTypeSet_\Sigma(H, \mathbf{A}, l, (\eta, \sigma')) \subseteq \hat{r}).$$

For algebras that preserve alias legality, the alias checking rules are sound.

$$[\text{Num}]\quad \Sigma; H \vdash \mathrm{N} : \texttt{Int} :: \{\} \qquad [\text{nothing}]\quad \Sigma; H \vdash \texttt{nothing} : \texttt{Void} :: \{\}$$

$$[\text{true}]\quad \Sigma; H \vdash \texttt{true} : \texttt{Bool} :: \{\} \qquad [\text{false}]\quad \Sigma; H \vdash \texttt{false} : \texttt{Bool} :: \{\}$$

$$[\text{ident}]\quad \Sigma; H \vdash \mathrm{I} : H(\mathrm{I}) :: H(\mathrm{I}) \qquad \text{if } \mathrm{I} \in dom(H)$$

$$[\text{call}]\quad \frac{\Sigma; H \vdash \vec{\mathrm{E}} : \vec{S} :: \vec{r}, \quad \Sigma.ResType(\mathrm{I}, \vec{S}) = T, \quad \Sigma.RetAlias(\mathrm{I}, \vec{S}, \vec{r}) = r'}{\Sigma; H \vdash \mathrm{I}(\vec{\mathrm{E}}) : T :: r'}$$

$$[\text{decl}]\quad \Sigma; H \vdash \mathrm{I} : \mathrm{T} \Longrightarrow [I \mapsto T]H$$

$$[\text{decl list}]\quad \frac{\Sigma; H \vdash \mathrm{D}_1 \Longrightarrow H', \quad \Sigma; H' \vdash \mathrm{D}_2 \Longrightarrow H''}{\Sigma; H \vdash \mathrm{D}_1 \,;\, \mathrm{D}_2 \Longrightarrow H''}$$

$$[\text{ExpCom}]\quad \frac{\Sigma; H \vdash \mathrm{E} : S :: r}{\Sigma; H \vdash \mathrm{E} \surd}$$

$$[\text{assign}]\quad \frac{\Sigma; H \vdash \mathrm{E} : S :: r, \quad H(I) = T, \quad S \leq \mathrm{T}, \quad r \subseteq \{\mathrm{T}\}}{\Sigma; H \vdash \mathrm{I} \texttt{ := } \mathrm{E} \surd}$$

$$[\text{Cond}]\quad \frac{\Sigma; H \vdash \mathrm{E} : \texttt{Bool} :: r, \quad \Sigma; H \vdash \mathrm{C}_1 \surd, \quad \Sigma; H \vdash \mathrm{C}_2 \surd}{\Sigma; H \vdash \texttt{if } \mathrm{E} \texttt{ then } \mathrm{C}_1 \texttt{ else } \mathrm{C}_2 \texttt{ fi} \surd}$$

$$[\text{Seq}]\quad \frac{\Sigma; H \vdash \mathrm{C}_1 \surd, \quad \Sigma; H \vdash \mathrm{C}_2 \surd}{\Sigma; H \vdash \mathrm{C}_1 \;;\; \mathrm{C}_2 \surd}$$

$$[\text{Main}]\quad \frac{\begin{array}{c}\Sigma; \{\} \vdash \mathrm{D}_1 \Longrightarrow H', \quad \Sigma; H' \vdash \mathrm{C}_1 \surd, \\ \Sigma; H' \vdash \mathrm{D}_2 \Longrightarrow H'', \quad checkVisible(\mathrm{D}_2), \quad \Sigma; H'' \vdash \mathrm{C}_2 \surd,\end{array}}{\Sigma \vdash \texttt{main observe } \mathrm{D}_1\ \mathrm{C}_1 \texttt{ by } \mathrm{D}_2\ \mathrm{C}_2 \surd}$$

Fig. 6. Type and alias checking rules for the main procedure part of INST.

Lemma 3.6 *Let M be a main procedure of INST. Let $\mathbf{A}$ be a Σ-algebra. If $\mathbf{A}$ preserves alias legality and $(H, s, f) = \mathcal{M}_\Sigma[\![M]\!]\mathbf{A}$, then $(\Sigma \vdash M \surd) \Rightarrow stAliasOk_\Sigma(H, \mathbf{A}, s)$.* □

4 Weak Behavioral Subtyping

The intuitive idea of behavioral subtyping is that each object of a subtype should behave like some object of its supertypes. One might think that to express "behaves like" it would be enough to simply relate abstract values. However, this would not take locations and hence aliasing into consideration. One cannot relate just locations either, because the abstract values stored in locations also determine behavior. Relating locations along with the store does not account for aliasing between variables in the environment. So one

must relate whole states. This idea is captured by the definition of simulation relations below.

4.1 Simulation Relations

The following formulation of simulation relations uses techniques from [13]. The bindable property ensures that simulation is preserved by assignments. The substitution property says that simulation relationships between states are preserved by method calls. It is expressed by assigning a variable to the value returned by the operations in each algebra, and then requiring that the resulting extended states be related. The coercion property is similar to the requirement that each object of a subtype should simulate some object of its supertypes. It ensures that each state simulates a state that does not use subtyping. The *EXTERNALS*-identical property says that a simulation relation is identity on values of visible types. This is used to compare the outputs of observations. The others are needed for technical reasons.

Definition 4.1 [simulation relation] Let $\mathbf{C}$ and $\mathbf{A}$ be Σ-algebras. A Σ-*simulation relation* $\mathcal{R}$ *from* $\mathbf{C}$ *to* $\mathbf{A}$ is a family of binary relations on states, $\langle \mathcal{R}_H : H \in TENV \rangle$, such that $\mathcal{R}_H \subseteq STATE_H[\mathbf{C}]_\perp \times STATE_H[\mathbf{A}]_\perp$ and for each type environment H each $(\eta_\mathbf{C}, \sigma_\mathbf{C}) \in STATE_H[\mathbf{C}]$, and each $(\eta_\mathbf{A}, \sigma_\mathbf{A}) \in STATE_H[\mathbf{A}]$, the following properties hold:

bindable: for each variable x, for each type T, and for each variable y such that $H(y) = T$, $l_y^\mathbf{C} = (\eta_\mathbf{C}\ y)$, and $l_y^\mathbf{A} = (\eta_\mathbf{A}\ y)$, if $\eta'_\mathbf{C} = [x \mapsto l_x^\mathbf{C}]\eta_\mathbf{C}$ and $\eta'_\mathbf{A} = [x \mapsto l_x^\mathbf{A}]\eta_\mathbf{A}$ then

$$(\eta_\mathbf{C}, \sigma_\mathbf{C})\ \mathcal{R}_H\ (\eta_\mathbf{A}, \sigma_\mathbf{A}) \Rightarrow (\eta'_\mathbf{C}, [l_x^\mathbf{C} \mapsto (\sigma_C\ l_y^\mathbf{C})]\sigma_\mathbf{C})\ \mathcal{R}_{[x \mapsto T]H}\ (\eta'_\mathbf{A}, [l_x^\mathbf{A} \mapsto (\sigma_A\ l_y^\mathbf{A})]\sigma_\mathbf{A})$$

substitution: for each tuple of types $\vec{S}$, for each type T, for each operation symbol $g : \vec{S} \to T$, for each tuple of variables $\vec{y}$ such that $H(\vec{y}) = \vec{S}$ and $\vec{v_\mathbf{C}} = (\sigma_\mathbf{C}\ (\eta_\mathbf{C}\ \vec{y}))$, $\vec{v_\mathbf{A}} = (\sigma_\mathbf{A}\ (\eta_\mathbf{A}\ \vec{y}))$, and for each variable x, if $\eta'_\mathbf{C} = [x \mapsto l_x^\mathbf{C}]\eta_\mathbf{C}$ and $\eta'_\mathbf{A} = [x \mapsto l_x^\mathbf{A}]\eta_\mathbf{A}$ then

$$\begin{aligned}(\eta_\mathbf{C}, \sigma_\mathbf{C})\ \mathcal{R}_H\ (\eta_\mathbf{A}, \sigma_\mathbf{A}) \Rightarrow\ & (\textbf{let}\ (r_\mathbf{C}, \sigma'_\mathbf{C}) = g^\mathbf{C}((\vec{v_\mathbf{C}}), \sigma_\mathbf{C})\ \textbf{in}\ (\eta'_\mathbf{C},\ [l_x^\mathbf{C} \mapsto r_\mathbf{C}]\sigma'_\mathbf{C})) \\ & \mathcal{R}_{[x \mapsto T]H} \\ & (\textbf{let}\ (r_\mathbf{A}, \sigma'_\mathbf{A}) = g^\mathbf{A}((\vec{v_\mathbf{A}}), \sigma_\mathbf{A})\ \textbf{in}\ (\eta'_\mathbf{A},\ [l_x^\mathbf{A} \mapsto r_\mathbf{A}]\sigma'_\mathbf{A})) \qquad (5)\end{aligned}$$

coercion: there exists a nominal state $(\eta'_A, \sigma'_A) \in STATE_H[\mathbf{A}]$, such that

$$(\eta_C, \sigma_C)\ \mathcal{R}_H\ (\eta'_A, \sigma'_A),$$

EXTERNALS-**identical:** for each type $T \in VIS$, for each variable x such that $H(x) = T$, if $(\eta_\mathbf{C}, \sigma_\mathbf{C})\ \mathcal{R}_H\ (\eta_\mathbf{A}, \sigma_\mathbf{A})$, then

$$externVal^\mathbf{C}(\sigma_\mathbf{C}(\eta_\mathbf{C}\ x), \sigma_\mathbf{C}) = externVal^\mathbf{A}(\sigma_\mathbf{A}(\eta_\mathbf{A}\ x), \sigma_\mathbf{A}),$$

shrinkable: if $H' \subseteq H$, $(\eta'_{\mathbf{C}}, \sigma'_{\mathbf{C}})$ and $(\eta'_{\mathbf{A}}, \sigma'_{\mathbf{A}})$ are H'-states, $(\eta'_{\mathbf{C}}, \sigma'_{\mathbf{C}}) \subseteq (\eta_{\mathbf{C}}, \sigma_{\mathbf{C}})$, and $(\eta'_{\mathbf{A}}, \sigma'_{\mathbf{A}}) \subseteq (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}})$, then

$$(\eta_{\mathbf{C}}, \sigma_{\mathbf{C}})\, \mathcal{R}_H\, (\eta_{\mathbf{A}}, \sigma_{\mathbf{A}}) \Rightarrow (\eta'_{\mathbf{C}}, \sigma'_{\mathbf{C}})\, \mathcal{R}_{H'}\, (\eta'_{\mathbf{A}}, \sigma'_{\mathbf{A}}),$$

bistrict: $\perp\, \mathcal{R}_H\, \perp$, and whenever $s\, \mathcal{R}_H\, s'$ and either s or s' is $\perp$, then so is the other.

Simulation relations preserve aliasing. That is, if two variables, x and y, are aliased in a state $s_{\mathbf{C}}$, and if $s_{\mathbf{C}}\, \mathcal{R}_H\, s_{\mathbf{A}}$, then x and y must be aliased in $s_{\mathbf{A}}$. If this were not the case, then one could observe changes in x by using operations on y in $s_{\mathbf{C}}$, while in $s_{\mathbf{A}}$ the same changes to y would not be observable through x. But this would violate the substitution property.

A careful reader might observe that the requirement that every state should be simulated by a nominal state in the "coercion" property eliminates certain kinds of direct aliasing. More precisely, it eliminates direct aliasing between variables of different types. The reason for this is the following. Suppose $S \neq T$, and consider a state in which and $x : T$ and $y : S$ were directly aliased. Then to satisfy the coercion property, such a state would have to be related to one where x and y both denoted objects of their types, and thus could not be directly aliased. This motivates the alias restrictions we impose on INST.

Note that the identity relation on states is not a Σ-simulation. A simulation relation also needs to have the coercion property.

Example 4.2 There is a Σ'-simulation relation from algebra $\mathbf{A}'$ (in Fig. 4 and Fig. 5) to itself.

A relation $\mathcal{R}' \subseteq STATE_H[\mathbf{A}'] \times STATE_H[\mathbf{A}']$ is defined such that $\perp\, \mathcal{R}'\, \perp$ and $(\eta_1, \sigma_1)\, \mathcal{R}'\, (\eta_2, \sigma_2)$ if and only if the following conditions hold:

- $(dom\ \eta_1) \subseteq (dom\ \eta_2)$
- For each type T, for each x:T $\in (dom\ \eta_1)$, if $v_1 = \sigma_1(\eta_1\ x)$, $v_2 = \sigma_2(\eta_2\ x)$, and $v_1 \in VALS_S^{\mathbf{A}'}$ then $v_2 = c(v_1, S, TtoS(T)^{\mathbf{A}'}, \sigma_1)$. The coercion function c is defined as follows.

$$c(v, S, S, \sigma) = v$$

$$c((v_0, l^{\texttt{MutablePair}}), \text{sortFor}(\texttt{MutableBSeq}), \text{sortFor}(\texttt{BoolSeq}), \sigma) =$$
$$\textbf{let}\ (v_1, v_2) = (\sigma\ l^{\texttt{MutablePair}})\ \textbf{in}$$
$$(v_0, v_1, v_2)$$

$$c((v_0, l^{\texttt{MutablePair}}), \text{sortFor}(\texttt{MutableBSeq}), \text{sortFor}(\texttt{StoreBool}), \sigma) =$$
$$\textbf{let}\ (v_1, v_2) = (\sigma\ l^{\texttt{MutablePair}})\ \textbf{in}$$
$$(v_0, v_1, v_2)$$

Then $\mathcal{R}'$ satisfies all the properties of a Σ'-simulation relation. □

However, there does not always exist a Σ-simulation relation from an algebra to itself. Consider an algebra, $\mathbf{B}'$, with the type `DestructBSeq` (a `BoolSeq`

with a destructive `update`) as a presumed subtype of `BoolSeq`. Then there cannot be a simulation relation from $\mathbf{B}'$ to itself because the operation `update` violates the substitution property.

4.2 Weak Behavioral Subtypes

The following definition of a weak behavioral subtype relation characterizes when a specification of several ADTs has a subtype relation ($\leq$) that is adequate for modular reasoning. Since we do not discuss the forms of type specifications, we use their denotations, which are sets of algebras that preserve alias legality.

Definition 4.3 [**weak behavioral subtyping**] Let *SPEC* be a set of Σ-algebras such that each $\mathbf{A}$ in *SPEC* preserves alias legality. The presumed subtype relationship $\leq$ on types (of Σ) is a *weak behavioral subtype relation for SPEC* if and only if for each $\mathbf{B} \in SPEC$ there is some $\mathbf{A} \in SPEC$ such that there is a Σ-simulation from $\mathbf{B}$ to $\mathbf{A}$.

If we let *SPEC* take all the Σ'-algebras isomorphic to $\mathbf{A}'$, then it is easy to see that the subtype relation of Σ' is a weak behavioral subtype relation. Hence this definition allows types with immutable objects, such as `BoolSeq`, to have subtypes with mutable objects, such as `MutableBSeq`.

Because this definition permits $\mathbf{B}$ and $\mathbf{A}$ to be different algebras, it works for incomplete specifications: those with observably different models. Such incomplete specifications are important in practice, so that a subtype can be more completely specified than its supertypes. (Unfortunately, space limitations do not allow us to give an example.)

Not every presumed subtype relation is a weak behavioral subtype relation, because of the coercion and substitution properties of simulation relations. For example, there is no weak behavioral subtype relation such that `DestructBSeq` is a subtype of `BoolSeq`.

4.3 Weak Behavioral Subtyping means No Surprises

We now show that the definition of weak behavioral subtyping is adequate for modular reasoning with supertype abstraction. We do this in a model-theoretic fashion, by first defining the set of expected results of an observation, or rather of a function from algebras to observations. The expected results are results of observations on nominal states – that is, states that do not use subtyping.

Definition 4.4 [**expected results**] Let *SPEC* be a set of Σ-algebras that preserve alias legality. Let H be a type environment. Let f be a function from Σ-algebras to H-observations. Then the set of *expected results of f for SPEC* is the union over all $\mathbf{A} \in SPEC$ and all $s_{\mathbf{A}} \in STATE_H[\mathbf{A}]$, such that $s_{\mathbf{A}}$ is nominal, of $(f\ \mathbf{A}\ s_{\mathbf{A}})$.

A result is *surprising* if it is not expected. Surprising results can occur if one uses a presumed subtype relation that does not satisfy the definition of

weak behavioral subtyping, and observes a state that is not nominal.

Theorem 4.5 (no surprises) *Let SPEC be a set of Σ-algebras that preserve alias legality. Let H be a type environment. Let $f : ALG \to OBS$ be such that there is some $\mathbf{A} \in SPEC$ and some main procedure M in INST such that $(H, s, f) = \mathcal{M}_\Sigma[\![M]\!]\ \mathbf{A}$.*

Then for all $\mathbf{C} \in SPEC$, and for all $s_\mathbf{C} \in STATE_H[\mathbf{C}]$, if $\leq$ is weak behavioral subtype relationship for SPEC, then $(f\ \mathbf{C}\ s_\mathbf{C})$ is an expected result for SPEC.

Proof (Sketch) Because $\leq$ is a weak behavioral subtype relation for *SPEC*, there is an $\mathbf{C}' \in SPEC$ and a Σ-simulation relation, $\mathcal{R}$, from $\mathbf{C}$ to $\mathbf{C}'$. Using structural induction, show that simulations are preserved by commands and declarations. Then in the semantics of the main procedure, the resulting states in the observation part ($(\eta''_\mathbf{B}, \sigma''_\mathbf{B})$ in the semantics of the main procedure) are related. So by the *EXTERNALS*-identical property, the resulting answer functions must give the same result for each variable (namely for those in D_2 of the main procedure M). □

The conclusion of the theorem does not hold if $\leq$ is not a weak behavioral subtype relation. The observation and the presumed subtype relation, between `DestructBSeq` and `BoolSeq`, in subsection 2.2 is an example of such a situation where unexpected results are observed because our presumed subtype relation is not a weak behavioral subtype relation.

The above theorem validates our definition of weak behavioral subtyping. It also computes the tasks mentioned in the introduction.

5 Related Work

Our work on the model-theory of behavioral subtyping is an extension of Leavens' work in [16]. Simulation relations in [16] relate only abstract values and hence cannot see any mutations in the state. Leavens and other model-theoretic approaches [3,14,12,15] do not deal with mutation and aliasing.

In contrast to our model-theoretic approach, America [1,2], and Liskov and Wing [18,17] give proof-theoretic definitions of behavioral subtyping. America does not deal with extra mutators in subtypes. Liskov and Wing allow extra mutators provided if the extra mutators can be explained in terms of the supertype methods or if they do not violate any history constraints. This rules out the possibility of mutable subtypes to immutable types. We leave for future work a direct comparison between our definition and such proof-theoretic definitions, and formulating the model-theoretic equivalent of Liskov and Wing's definitions.

The subtype relationships between various collection types in Cook's hierarchy [8] are weak behavioral subtypes in our sense. For immutable record types, our definition matches Cardelli's rules [4]. The type hierarchies discussed in [17] are weak behavioral subtypes.

6 Discussion

The most interesting weak behavioral subtypes are between mutable and immutable types. For example, a mutable type, `Array[Int]` can be specified as a weak behavioral subtype of an immutable type `Sequence[Int]`. This allows, for example, a procedure that computes the sum of a `Sequence[Int]` to be applied to an `Array[Int]` object. Similarly, a mutable record type can be specified as a subtype of an immutable record type with fewer fields.

One can even have a hierarchy of weak behavioral subtypes, with increasing degrees of mutability. As an example, a completely mutable array is a weak behavioral subtype of partially mutable array, which in turn is a weak behavioral subtype of an immutable array.

Subtype objects can have more state than supertype objects. A `Triple` can be a weak behavioral subtype of `Pair` with the only requirement that the degree of mutability of `Triple` should be at least equal to that of the `Pair`. That is, if the first component of the `Pair` can be mutated, then the first component of the `Triple` should also be mutated but there is no constraint on the mutability of the other components of a `Triple`.

For weak behavioral subtyping to be adequate for supertype abstraction, one needs to prohibit direct aliases between objects of different types. We suggested a way to use multi-method dispatch to avoid part of the burden this places on expressiveness, by having the programmer define enough methods.

The burden of our aliasing restrictions can be weakened still further by allowing direct aliasing between variables of immutable types. Finally if a programming language supported both our notion of weak behavioral subtyping, and Liskov and Wing's strong behavioral subtyping, direct aliasing could be allowed between strong behavioral subtypes.

We do however, allow aliasing between objects of the same type and indirect aliasing between objects of different types. An `MutableBSeq` object is indirectly aliased to a `MutablePair` object in the `mkMutableBSeq` method in Fig. 2.

7 Summary

The main contribution of our work is a new definition of subtyping for arbitrary deterministic abstract data types in the presence of mutation and aliasing. This definition is weaker than Liskov and Wing's definitions [18,17], because it allows types with immutable objects to have subtypes with mutable objects. This flexibility seems to be important in practice. The price to be paid, however, is that the language must restrict aliasing. We have given suitable aliasing restrictions, which disallow direct aliasing between identifiers of different types. We believe that such aliasing restrictions may actually be of some practical benefit, as they allow naive reasoning to be sound.

Acknowledgements

Thanks to Ian Maung discussions about modular reasoning and to Don Pigozzi, Barbara Liskov, Jeannette Wing, and the participants at the 1993 *Foundations of OO Languages* workshop for discussions about our work on this topic. Thanks to John Mitchell for his suggestion that we should present more of the language, which helped clarify this work. Thanks also to Luca Cardelli and to the anonymous referees, for their suggestions, which helped improve the paper.

References

[1] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, N.Y., June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[2] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[3] Kim B. Bruce and Peter Wegner. An algebraic model of subtype and inheritance. In Francois Bançilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, pages 75–96. Addison-Wesley, Reading, Mass., August 1990.

[4] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, N.Y., 1991.

[5] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, N.Y., 1992.

[6] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. *ACM SIGPLAN Notices*, 29(10):1–15, October 1994. OOPSLA '94 Conference Proceedings, October 1994, Portland, Oregon.

[7] Jolly Chen. The Larch/Generic interface language. Technical report, Massachusetts Institute of Technology, EECS department, May 1989. The author's Bachelor's thesis. Available from John Guttag at MIT (guttag@lcs.mit.edu).

[8] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. *ACM SIGPLAN Notices*, 27(10):1–15, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[9] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[10] Gary T. Leavens and Krishna Kishore Dhara. Blended algebraic and denotational semantics for ADT languages. Technical Report 93-21b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 1994. Submitted for publication. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[11] Gary T. Leavens and Don Pigozzi. Typed homomorphic relations extended with subtypes. Technical Report 91-14, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, June 1991. Appears in the proceedings of *Mathematical Foundations of Programming Semantics '91*, Springer-Verlag, Lecture Notes in Computer Science, volume 598, pages 144-167, 1992.

[12] Gary T. Leavens and Don Pigozzi. Typed homomorphic relations extended with subtypes. In Stephen Brookes, editor, *Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 144–167. Springer-Verlag, New York, N.Y., 1992.

[13] Gary T. Leavens and Don Pigozzi. The behavior-realization adjunction and generalized homomorphic relations. Technical Report 94-18, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 1994. Submitted for publication.

[14] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). *ACM SIGPLAN Notices*, 25(10):212–223, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[15] Gary T. Leavens and William E. Weihl. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report 92-28d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, August 1994. Full version of a paper to appear in *Acta Informatica*. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[16] Gary Todd Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1989. The author's Ph.D. thesis.

[17] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[18] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[19] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, January 1980.

[20] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.